ELSEVIER

Science of Computer Programming 39 (2001) 149–188

Science of Computer Programming

www.elsevier.nl/locate/scico

Mechanisms for naming
An algebraic approach with an application to Java

Loe M.G. Feijs *

Eindhoven University of Technology, HG7.33, P.O. Box 513, 5600MB, Netherlands

Received 5 July 1999

Abstract

The present paper investigates the hypothesis that a variety of mechanisms for naming can be understood as algebraic concepts. These concepts are developed and then they are applied to aspects of Java to see whether indeed they lead to compact characterizations of the language's mechanisms for naming. Focus is on object oriented themes: inheritance, polymorphism and encapsulation. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Java; Inheritance; Type theory

1. Introduction

1.1. Survey

Mechanisms for naming are important in the design of programming languages and specification languages where identifiers are being declared and used for types, variables, functions, etc. They are equally important in distributed systems where addresses, unique-object identifiers, uniform resource locators, etc. have to be traced back to get the objects they refer to via indirection tables, name servers, etc. Much effort on the study of languages has gone into semantic issues, such as recursion, proof rules and type theory, whereas the mechanisms for naming are considered less interesting, and being of 'only' a syntactic nature. We believe that the subject deserves more attention and study; there are options for developing fundamental and useful theory, techniques and tools that help in choosing and implementing mechanisms for naming. The present paper is meant as a contribution to this study. We develop and analyse several mechanisms, illustrating them with examples about scoping and overloading resolution in Java

* Tel.: +31-40-2475020.
E-mail address: feijs@win.tue.nl (L.M.G. Feijs).

PII: S0167-6423(00)00003-4

(although the mechanisms for naming are much wider applicable, we think it helps to be concrete with respect to examples, and that is why we choose Java here [3]).

We start with primitive concepts of names, types and semantic elements, elements for short. The main objects of study are relations, sometimes mappings,

- from names to elements,
- from names to types,
- from names to pairs ⟨type, element⟩.

We call these *denotations*, *typings* and *overloadings*, respectively. The term 'denotation' is chosen because a name *denotes* a semantical element. The reader may think of overloadings as a kind of 'symbol tables'. We should give an account why we chose the term 'overloading'. It refers to the special symbol tables in which a name can occur more than once because the programming language for whose compiler the symbol table is meant, supports the language feature usually called 'overloading'. The language feature called 'overloading' is explained in almost any tutorial on an object-oriented programming language (it means that it is allowed for more than one function to have the same name). For denotations, several algebraic operators suggest themselves readily: denotations can be added in an overriding and in a non-overriding way. Along these lines we can describe how the set of declarations of one scope is to be combined with the set of declarations of another scope.

When names are used, this is often in tree-like terms, such as `f(c)` or `c.f` which makes sense if we have a denotation δ which maps `f` to a function and `c` to an element in that function's domain. In algebraic terms, we must study how a denotation δ is extended to a mapping δ' which maps tree-like terms to semantic elements. Similarly, a typing τ is extended to a mapping τ' which maps tree-like terms to types where the latter mapping is partial.

Instead of denotations or typings, overloadings can be used, which are combined semantics- and type-oriented mappings. We can study the same kind of operations and extensions, but there are a number of complications to be dealt with (notably overloading resolution). Resolution is the process of transforming a typing, τ say, into a functional mapping τ' from tree-like terms to elements, reconstructing a unique type for each name-occurrence (in the same way resolution is necessary for overloadings). [1] Additional complications are introduced by inheritance and subtyping.

The paper is organized as follows: in Section 1.2 we give a motivating example in Java. Then in Section 2 we summarize some mathematical tools, most of which are well-known. We use these tools in Sections 3–5 to define a number of more complicated operators, which are directly applicable in the sense that each of them models a particular mechanism for naming, such as a way of combining namings or a rule of doing overloading resolution. We call the latter operators 'constructions',

[1] We use the term 'resolution' rather than type inference because most often the term 'type inference' is used for much complexer type systems such as the Hindley–Milner system for which type inference indeed is closely related to inferencing and reasoning in (propositional) logic. The reader may think of 'resolution' as *type inference for languages with the overloading feature*.

because we have a certain constructive style of describing them. In Sections 3–5 we propose constructions to model aspects of inheritance, polymorphism and encapsulation, respectively. Finally, in Section 6 a number of concluding remarks are given.

1.2. Motivating example

Computer programs are represented as texts containing names, next to keywords, bracketing constructs and operators, etc. Alternatively, programs can be viewed as trees where names appear as leaves and where the language-specific ingredients label the various kinds of branching nodes. In either case, the occurrences of names can be classified into *defining* and *applied* occurrences. The same applies to passive object-definitions, e.g. pages of the world-wide web associated with defining occurrences of URLs; each page may contain references to other URLs. When the program (or object) is processed, by compilation or interpretation, it is necessary to analyse the applied occurrences, tracing them back to their definitions.

The example below exhibits three technicalities playing a role when tracing back the definitions for given applied occurrences. It is a toy-example of Java text about types of coffee and coffee-with-latte. We add some explanation already: the functions of type `boolean ok(boolean am)` check the liquid-parameters for a given time of the day. During AM, it is important that the coffee is black (blackness $\geqslant 10$, even more when milk is present). In the afternoon, residuals must be low.

```
class Coffee {
   public int b; // blackness
   public int r; // residuals

   public boolean ok(boolean am) {
      if (am)  return (b > 10);
         else  return (r < 2 );
}  }

class Latte extends Coffee {
   public int m; // milk

   public boolean ok(boolean am) {
      if (am)  return (b > 10+m);
         else  return (r < 3  ) ;
   }

   public boolean ok(int t) {
      if (t < 12) return (b > 10+m);
         else     return (r <  3  );
   }
```

```
      public boolean test(int t) {
         if (ok(false)) return (not ok(t));
            else        return true       ;
   }  }

   class Drinker {
      Latte l;

      int drink(int t) {
         boolean d;
         d = l.test(t);
   }  }
```

We describe the above-mentioned 'technicalities' now. The first is related to inheritance, that is the `extends` construct of Java, being one of the occasions where namings are combined. The next two technicalities are related to polymorphism [2] and encapsulation, respectively.

- the name `b` as occurring in `b>10+m` inside the definition of `ok(boolean am)` for `Latte` refers to a field `int b` which belongs to `Latte`. This is the case because `Latte` extends `Coffee` and because `Coffee` has a field `int b`. So `extends` combines `Coffee`'s naming with an additional naming built-up in the class `Latte`. Something similar happens for functions, but the new `ok(boolean am)` overwrites the earlier `ok(boolean am)` of `Coffee`, whereas the function `ok(int t)` can co-exist next to `ok(boolean am)` without overwriting or clashing.
- one identifier '`ok`' is used for distinct functions, and its interpretation is dependent on the context of a given applied occurrence. So, a statement `if (ok(false)) return (not ok(t))` can be analysed on the basis of the typing system to find out that the first occurrence must be the `ok(boolean am)` of `Latte` whereas the second occurrence is the `ok(int t)`. This analysis process is called resolution. [3]
- the function `ok(boolean am)` as defined in `Latte()` can be invoked within the scope of `Latte` just by writing `ok(true)`. When this function is to be used in *another* class, it has to be invoked by writing `l.ok(i)` where `l` refers to an object of type `Latte`. For the same reason we see the invocation `d=l.test(t)` in `drink` in `Drinker`; writing `d=test(t)` at that place would be an error.

Now the central question of this paper is: how can these technicalities be explained when adopting the idea that namings (i.e. denotations, typings and overloadings) are

[2] In other articles, the word 'polymorphism' is also used for the phenomenon of a term expecting explicit or implicit type arguments. For example a function on lists being polymorphic in the type of list-elements (as possible in the Hindley–Milner system). In the object-oriented community however, where no Hindley–Milner systems are used, the word 'polymorphism' applies to the situation of two distinct functions with the same name, as in our article.

[3] Of course this is an example of the matter mentioned in Section 1 where we announced that we shall study ways in which a denotation δ is extended to a mapping δ' which maps tree-like terms to semantic elements and similarly for 'overloadings'.

subject to manipulation by algebraic operators? Which are these operators? In the next sections we set out to identify them.

2. Mathematical tools

In this section we present a number of mathematical tools, some of which based on a modest amount of set theory and a few commuting diagrams. We begin with a short section on the methodology and on aspects of the mathematics in the present article.

2.1. Methodological remarks

The main question we would like to address in this subsection is how much mathematics is it worth to have really compact characterisations of a language's mechanisms for naming? The reader is offered many pages of mathematics in the present paper and only in Sections 3.3, 3.5, 3.6 and in Section 4.6 (efficient comparison of language styles) there is a certain reward, next to the examples in (the last paragraphs of) Sections 4.2–4.5.

What is the nature and the difficulty of the proposed mathematics? Essentially we only start from basic things such as sets, Cartesian products, and elementary properties of mappings such as injectivity, functionality and surjectivity. Some of the notations and conventions to be developed in Section 2.2 are not standard, but they are developed precisely for the purpose of arriving at short and compact notations. We use some commuting diagrams but we do not really use category theory.

The goal of the paper is to arrive at characterisations of language constructs using algebraic means. Perhaps the idea of *algebra* looks as a kind of overkill, especially since some of the operators involved are probably not widely known (yet). But this should be compared to other branches of science where algebra has proven its value: when studying symmetry in the early days *before* group theory, almost anyone interested in crystals or tilings would probably have considered proposals for formulating group theoretic laws as a difficult and theoretical exercise (at present we are familiar with group theory and we do not consider the group laws to be difficult after all). As a more recent example, consider the advent of relational database theory, proposed by Codd and others. This required investments in algebraic theory as well, but once understood, it certainly had a tremendous impact. We do not want to say that the present article will have such an impact (we cannot know that). We argue that trying to understand something by means of algebra is a way of trying to make progress.

Let us consider in how far we will get really compact characterisations of a language's mechanisms for naming. The reader is kindly requested to have a quick preview at, e.g. Figs. 1–5. The figures depend on the definition of the operations such as $+_1$, $+_{1\frac{1}{2}}$, $+_2$, $\dot{+}_1$, etc. occurring in the figures. The figures can be viewed as accurate and compact expressions of a number of relevant scope rules. Not only do they show how things work in Java, but they also show how the language could have been

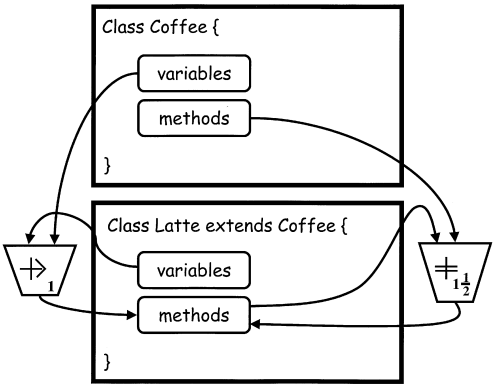

Fig. 1. Inheritance and typings.

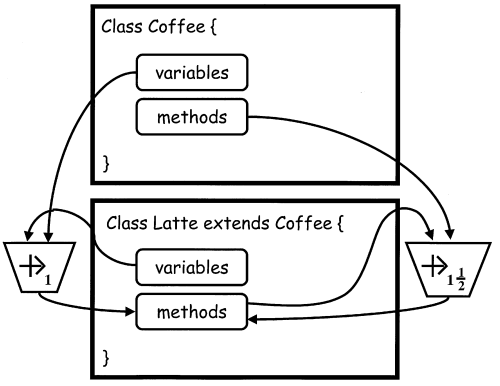

Fig. 2. Inheritance and overloadings.

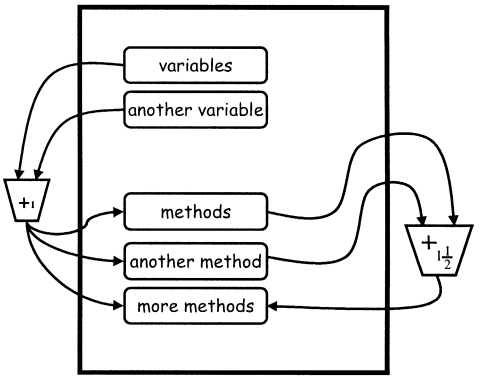

Fig. 3. Scoping and overloadings.

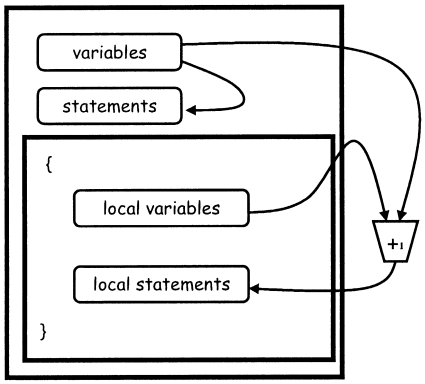

Fig. 4. Scoping and overloadings.

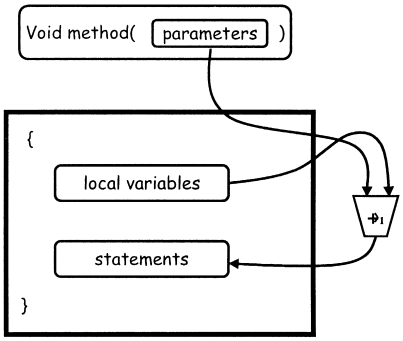

Fig. 5. Parameter scoping and overloadings.

designed differently (for example, Java's language designer could have chosen to use $+_1$ in Fig. 5 instead of $+_1$).

It is also possible to compare the language characterisations with the work of a compiler (in Section 3.3 we shall make such a remark). But it would not be fair to say that we need not look for a better understanding of Java and related languages because we can always ask our compiler what is correct and what is not. From a short-term practical point of view this may be satisfactory, but from a scientific point of view it is not. We want to understand the language without reference to the compiler (the compiler implements the language, it should not define it). Modern compiler code is much more readable than old hand-crafted compilers because of modern definition tools such as attribute grammars and type theory (indeed we aim at making another contribution to that tool box).

2.2. Views on sets

In connection with overloadings, we must prepare ourselves for manipulating ternary relations. We must carefully distinguish product sets such as $A \times (B \times C)$ and $(A \times B) \times C$.

For example, if we say that $r \subseteq A \times (B \times C)$ is functional, then this means that for each $a \in A$ there is at most one pair $\langle b, c \rangle$ with $b \in B$ and $c \in C$ such that the pair $\langle a, \langle b, c \rangle \rangle$ is in r (usually we denote that pair as a maplet $a \mapsto \langle b, c \rangle$). But if we say that $r \subseteq (A \times B) \times C$ is functional, then this means that for each pair $\langle a, b \rangle$, $a \in A$ and $b \in B$, there is at most one $c \in C$ such that $\langle \langle a, b \rangle, c \rangle \in r$ (or, which denotes the same, $\langle a, b \rangle \mapsto c$).

Of course $(A \times B) \times C$ is equivalent to $A \times (B \times C)$ in the sense that there is a bijective (= one–one mapping) between these two sets. This situation is expressed as

$$(A \times B) \times C \cong A \times (B \times C).$$

For certain purposes it makes a difference whether we consider a relation r as a subset of $(A \times B) \times C$, or whether we consider the corresponding relation on $A \times (B \times C)$. In the former case we say that we view r as a relation in two arguments and in the latter case as a relation in one argument. This saying is particularly intuitive if r is functional. Note that r being functional in one argument implies that r can be viewed as a functional relation in two arguments, but not conversely. We develop some mathematical notation that will be of help when dealing with this.

If we are given two sets, A and B we write $A \cong B$ if there is a bijective mapping from A to B. For example,[4] $\mathscr{P}(A \times B) \cong (A \to \mathscr{P}(B))$ since we have the mapping α given by $\alpha : r \mapsto f$ where f is given by $f(a) = \{b \in B \mid arb\}$ for all $a \in A$. If we want to make the mapping explicit, we attach it to the $\cong$ symbol, such that for example, $\mathscr{P}(A \times B) \cong_\alpha A \to \mathscr{P}(B)$ for the α just given. In many cases we shall not spell out the definition of the bijective mapping explicitly, because it is considered obvious (for example it may suffice to say that it is re-bracketing of a Cartesian product). Let $A \cong_\alpha B$, then we write $a \sim_\alpha b$ if $\alpha(a) = b$. In Section 2.3 we shall adopt a convention that restricts the mappings α to a specific class of mappings.

Next, we shall define several generalizations of the concept of functionality. These will be denoted as functional$_1$, functional$_2$ and functional$_{1\frac{1}{2}}$. Before giving the definitions we add a note on notation: next to identifiers r, τ, a, ω, etc., we also use $r_{(1)}$, $r_{(2)}$, $r_{(1\frac{1}{2})}$, etc., as identifiers (so *bracketed index* is not an operator), usually in a context like for example $r \sim_\alpha r_{(1)}$ or $r \sim_\beta r_{(1\frac{1}{2})}$. In the same way we use $\tau_{(1)}$, $\tau_{(1\frac{1}{2})}$, etc.

If we have a relation $r \in \mathscr{P}(A \times (B \times C))$ we say that r is functional$_1$ if r is functional.[5] If we have a bijective mapping α by which $\mathscr{P}(A \times (B \times C)) \cong_\alpha \mathscr{P}((A \times B) \times C)$, then we define that our relation $r \in \mathscr{P}(A \times (B \times C))$ is functional$_2$ if $r \sim_\alpha r_{(2)}$ where $r_{(2)}$ is functional. If we have a bijective mapping β by which $\mathscr{P}(A \times ((B_1 \times B_2) \times C)) \cong_\beta \mathscr{P}((A \times B_1) \times (B_2 \times C))$ and if moreover B happens to be a product, say $B = B_1 \times B_2$, then we say that r is functional$_{1\frac{1}{2}}$ if $r \sim_\beta r_{(1\frac{1}{2})}$ and $r_{(1\frac{1}{2})}$ is functional.

[4] We use $\mathscr{P}$ to denote 'powerset', as usual.

[5] A relation r is functional if each element in the domain is r-related to at most one element in the range; in other words, it must be a function.

Table 1

Name	⟨function type,function body⟩
ok	⟨⟨boolean,boolean⟩, f_1⟩
ok	⟨⟨int,boolean⟩, f_2⟩
test	⟨⟨int,boolean⟩, f_3⟩

Table 2

⟨name,function type⟩	function body
⟨ok,⟨boolean,boolean⟩⟩	f_1
⟨ok,⟨int,boolean⟩⟩	f_2
⟨test,⟨int,boolean⟩⟩	f_3

Table 3

⟨name,type⟩	⟨type,function body⟩
⟨ok,boolean⟩	⟨boolean, f_1⟩
⟨ok,int⟩	⟨boolean, f_2⟩
⟨test,int⟩	⟨boolean, f_3⟩

Example. Let A be a set of names including `ok` and `test`, B a set of pairs of types such as ⟨`boolean,boolean`⟩ and ⟨`int,boolean`⟩, and let C be a set of function bodies denoted as f_1, f_2 and f_3. A pair of types represents a function type, viz. its input type and its output type. We think of f_1, f_2 and f_3 as the three function bodies of `Latte` of Section 1.2. Let ω be the relation from names to pairs ⟨function type,function body⟩ as given by Table 1.

This is an example of what we called an 'overloading' in Section 1.1. We find that this ω is not functional$_1$ because it is not functional (for `ok` there is not a unique pair ⟨function type, function body⟩, there are *two* pairs). But using the bijective mapping α by which $\mathscr{P}(A \times (B \times C)) \cong_\alpha \mathscr{P}((A \times B) \times C)$ we find that $\omega \sim_\alpha \omega_{(2)}$ where $\omega_{(2)}$ is given by Table 2.

Because the relation of the latter table is functional we may conclude that ω is functional$_2$. Next, we note that B happens to be a product of the form $B_1 \times B_2$ so we may refer to the bijective mapping β by which $\mathscr{P}(A \times ((B_1 \times B_2) \times C)) \cong_\beta \mathscr{P}((A \times B_1) \times (B_2 \times C))$. Using this we see that $\omega \sim_\beta \omega_{(1\frac{1}{2})}$ if we take for $\omega_{(1\frac{1}{2})}$ the mapping of Table 3.

Again, the relation of the latter table is functional and hence we may conclude that that ω is functional$_{1\frac{1}{2}}$. (end of example).

We note that each α, β, etc., defines a transformation whose effect can be undone and therefore we shall sometimes refer to other well-known transformations groups, notably rotation groups, using them as a metaphor. We shall talk about the transition from ω to $\omega_{(1\frac{1}{2})}$ as if it were a rotation, saying that we rotate ω into its $\omega_{(1\frac{1}{2})}$ position.

As a preparation for Section 2.3 we introduce some further notations. First, $f : A \rightharpoonup B$ means that f is a partial function from A to B. Next, $\mathscr{P}(B)$ is the set of all subsets of

B, whereas $\mathscr{P}_{\leqslant 1}(B)$ is the set of all subsets of B which contain at most one element. Similarly, $\mathscr{P}_{=1}(B)$ is the set of all subsets of B which contain precisely one element. Formally $\mathscr{P}(A)=\{s \mid s \subseteq A\}$ and $\mathscr{P}_{\leqslant 1}(A)=\{s \mid s \subseteq A \wedge |s| \leqslant 1\}$, or which is the same, $\mathscr{P}_{\leqslant 1}(A)=\{\emptyset\} \cup \{\{x\} \mid x \in A\}$. As usual, a *multifunction* is a function whose range is a powerset, for example $f : A \to \mathscr{P}(B)$, $g : A \to \mathscr{P}_{\leqslant 1}(B)$ or $h : A \to \mathscr{P}_{=1}(B)$.

2.3. Canonical transformations

From Section 3 onwards we shall engage in a constructive activity where we frequently have to do conversions between various representations of certain sets. For that purpose we adopt the convention that $\subseteq$ symbols (meaning set-inclusion) and unsubscripted $\cong$ symbols (meaning the existence of a bijective mapping), unless explicitly told otherwise, only refer to a restricted class of equivalences and embeddings between sets, viz. those which are generated by:

- Currying and un-Currying,
- associativity of Cartesian product formation,
- transformation to/from multifunctions,
- inclusion between $\mathscr{P}_{=1}(\ )$, $\mathscr{P}_{\leqslant 1}(\ )$ and $\mathscr{P}(\ )$.

We call them *canonical* transformations. We shall explain each of the canonical transformations in more detail now.

By *Currying* we mean transforming a function r of two arguments, say $r : A \times B \to C$, into a function $r_{(c)} : A \to (B \to C)$ which takes its arguments one by one; the transformation is such that for all $a \in A$ and $b \in B$ we find that $r_{(c)}(a)(b) = r(\langle a, b \rangle)$. See for example [12, p. 2], or [4, p. 6], where the idea is attributed to Schönfinkel [15].

By *associativity of Cartesian product formation* we refer to the obvious rebracketing $A \times (B \times C) \cong (A \times B) \times C$.

By *transforming to multifunctions* we refer to $\mathscr{P}(A \times B) \cong (A \to \mathscr{P}(B))$ because of the mapping $r \mapsto f$ for $r \subseteq A \times B$ and $f : A \to \mathscr{P}(B)$ given by $f(a) = \{b \in B \mid arb\}$ for all $a \in A$. There are two related transformations to multifunctions which we allow as well; the first of these, dealing with partial functions, is the transformation based on $(A \rightharpoonup B) \cong (A \to \mathscr{P}_{\leqslant 1}(B))$ because of the mapping $f \mapsto g$ where g is given by $g(a) = \{f(a)\}$ if $f(a)$ is defined, $g(a) = \emptyset$ otherwise. The second of these goes analogously, viz. $(A \to B) \cong (A \to \mathscr{P}_{=1}(B))$.

For the subsets of $A \times B$, the canonical embeddings involved are summarized by the diagram

$$\begin{array}{ccc}
(A \to B) & \cong & A \to \mathscr{P}_{=1}(B) \\
\downarrow\subseteq & & \downarrow\subseteq \\
A \rightharpoonup B & \cong & A \to \mathscr{P}_{\leqslant 1}(B) \\
\downarrow\subseteq & & \downarrow\subseteq \\
\mathscr{P}(A \times B) & \cong & A \to \mathscr{P}(B)
\end{array}$$

Each row in this diagram is an instance of a tranformation to multifunctions. Each column represents two inclusion statements, which are easily verified; for example $(A \to B) \subseteq (A \rightharpoonup B)$ because each function $f : A \to B$ is a special case of a partial function, but not the other way around.

We give a further illustration by considering the subsets of $(A \times B) \times C$; the equivalences involved are those of the following diagram together with the transformations that may occur nested inside A, B or C.

$$\begin{array}{ccc} \mathscr{P}((A \times B) \times C) & \cong & \mathscr{P}(A \times (B \times C)) \\ \updownarrow\cong & & \updownarrow\cong \\ (A \times B) \to \mathscr{P}(C) & \cong & A \to \mathscr{P}(B \times C) \\ \nwarrow\!\!\searrow\cong & & \swarrow\!\!\nearrow\cong \\ & A \to (B \to \mathscr{P}(C)) & \end{array}$$

For each given relation, these transformations form a (partial) group, since each pair of applicable transformations can be composed to form another transformation and each transformation step can be undone. The partial transformation subgroup of the associativity rule alone has already a kind fractal structure (options for re-bracketing are inside bracketed expressions and inside bracketed expressions inside bracketed expressions etc.), upto a certain depth which is given by the complexity of the relation being considered.

As a consequence of the above conventions, we do *not* allow writing for example $\mathscr{P}(A \times B) \cong \mathscr{P}(B \times A)$ for general A and B, although a bijection exists.

2.4. Enforcing uniqueness

The present Subsection describes the first of our 'mathematical tools' (it is useful, among other things, for resolution). Each set A gives rise to a powerset $\mathscr{P}(A)$. Converting an element of $\mathscr{P}(A)$ into an element of $\mathscr{P}_{\leqslant 1}(A)$ implies some loss of information; we may employ the 'uniqueness operator' $\mathscr{U} : \mathscr{P}(A) \to \mathscr{P}_{\leqslant 1}(A)$ given by

$$\mathscr{U}(s) = \begin{cases} s & \text{if } |s| = 1, \\ \emptyset & \text{otherwise,} \end{cases}$$

where $|s|$ denotes the number of elements in the set s. It is easy to turn a non-functional relation into a partial function

$$\begin{array}{c} \mathscr{P}(A \times B) \\ \downarrow\cong \\ A \to \mathscr{P}(B) \\ \downarrow\forall\mathscr{U} \\ A \to \mathscr{P}_{\leqslant 1}(B) \\ \downarrow\cong \\ A \rightharpoonup B. \end{array}$$

We add some explanation on the above-mentioned $\forall$ functor. It is defined by the requirement that $(\forall \mathcal{O})(f)(a) = \mathcal{O}(f(a))$ hold for an arbitrary operator $\mathcal{O} : X \to Y$ and for all $a \in A$. In the above-mentioned case we take $\mathcal{P}(B)$ for X and $\mathcal{P}_{\leqslant 1}(B)$ for Y.

2.5. Adding partial functions

If two relations $r_1, r_2 \subseteq A \times B$ happen to be functional, so $r_1, r_2 : A \rightharpoonup B$, it is interesting to combine them to get an $r_3 \subseteq A \times B$ in such a way that r_3 is functional too. This plays a role in Java when combining the declarations of one scope with the operations of another scope, for example when nesting scopes by means of { and } or when extending a class.

We seek to choose a suitable operation for such 'combine' action, and for the time being we shall denote it as $+?$. To illustrate the difficulty involved, consider

$$\frac{\begin{array}{l}\{\ a_1 \mapsto b_1,\ a_2 \mapsto b_2, \qquad\quad\ \}\\ \{\ a_1 \mapsto b_3, \qquad\quad a_3 \mapsto b_4\ \}\end{array}}{\{\ a_1 \mapsto ?,\ a_2 \mapsto b_2,\ a_3 \mapsto b_4\ \}}\ +?$$

The difficulty can be resolved in three distinct ways. The first solution is to give priority to the maplets of the addition's first argument: the conflict between $a_1 \mapsto b_1$ and $a_1 \mapsto b_3$ is resolved by keeping the former and ignoring the latter. We denote this addition as $\dot{+}$. We let $\dot{+}$ be a total operation.[6]

Definition ($\dot{+}$)**.** Let r_1 and r_2 be functional. Then $\dot{+} : (A \rightharpoonup B) \times (A \rightharpoonup B) \to (A \rightharpoonup B)$ is defined as follows: $(a \mapsto b) \in (r_1 \dot{+} r_2)$ iff $(a \mapsto b) \in r_1$ or $a \notin \mathrm{dom}(r_1) \wedge (a \mapsto b) \in r_2$.

Note that $\dot{+}$ is a total operation. Using the notational device of function application

$$(r_1 \dot{+} r_2)(a) = \begin{cases} r_1(a) & \text{if defined,} \\ r_2(a) & \text{otherwise.} \end{cases}$$

The second solution is not to choose among the maplets $a_1 \mapsto b_1$ and $a_1 \mapsto b_3$, but to stipulate that in this case the addition is not defined. So $+$ is a partial operator.

Definition ($+$)**.** Let r_1 and r_2 be functional. Then $+ : (A \rightharpoonup B) \times (A \rightharpoonup B) \rightharpoonup (A \rightharpoonup B)$ is defined as follows: $r_1 + r_2$ is defined if for no a we find that both $r_1(a)$ and $r_2(a)$ are defined. In that case, $(a \mapsto b) \in (r_1 + r_2)$ iff $(a \mapsto b) \in r_1$ or $(a \mapsto b) \in r_2$.

Using the notational device of function application, assuming that $r_1 + r_2$ is defined, then $(r_1 + r_2)(a) = r_1(a)$ if defined, otherwise $r_2(a)$ if defined, but 'undefined' if both undefined.

[6] We call it preferential addition, using the symbol $\dot{+}$ earlier used in [8] where it was attempted to model TTCN-inspired language constructs in process algebra. In the latter context, the symbol $\dot{+}$ was advised to the author by J.A. Bergstra, who suggested to call it preferential choice.

The third solution is to avoid choosing among the maplets $a_1 \mapsto b_1$ and $a_1 \mapsto b_3$ by demanding that b_1 equals b_3. We denote this addition as $\dotplus$. In other words, $\dotplus$ is a partial operation.[7]

Definition ($\dotplus$)**.** Let r_1 and r_2 be functional. Then $\dotplus : (A \rightharpoonup B) \times (A \rightharpoonup B) \rightharpoonup (A \rightharpoonup B)$ is defined as follows: $r_1 \dotplus r_2$ is defined if for no a we find that $r_1(a)$ is defined and $r_2(a)$ is defined and $r_1(a) \neq r_2(a)$. When defined, $(a \mapsto b) \in (r_1 \dotplus r_2)$ iff $(a \mapsto b) \in r_1$ or $(a \mapsto b) \in r_2$.

The idea is that r_1 and r_2 must agree on the common part of their domains. If they do not agree, then the addition $r_1 \dotplus r_2$ is undefined. If they agree, then the addition $r_1 \dotplus r_2$ is defined (although it may still happen that for specific values of a we find that both $r_1(a)$ and $r_2(a)$ are undefined and therefore $(r_1 \dotplus r_2)(a)$ is undefined).

We checked for elementary algebraic properties. Since the laws are conditional equations we need a notation for definedness[8] for which we choose '!'.

Proposition. *For functional relations* p, p_1, p_2, p_3

$$
\begin{aligned}
&p \dotplus \emptyset = p,\\
&\emptyset \dotplus p = p,\\
&p \dotplus p = p,\\
&(p_1 \dotplus p_2) \dotplus p_3 = p_1 \dotplus (p_2 \dotplus p_3),\\
&(p_1 \dotplus p_2) \dotplus p_1 = p_1 \dotplus p_2,
\end{aligned}
$$

$$
\begin{aligned}
&p + \emptyset = p,\\
&\emptyset + p = p,\\
&(p + p)! \Rightarrow p = \emptyset,\\
&(p_1 + p_2)! \Rightarrow p_1 + p_2 = p_2 + p_1,\\
&((p_1 + p_2) + p_3)! \Rightarrow (p_1 + p_2) + p_3 = p_1 + (p_2 + p_3),
\end{aligned}
$$

$$
\begin{aligned}
&p \dotplus \emptyset = p,\\
&\emptyset \dotplus p = p,\\
&(p \dotplus p) = p,\\
&(p_1 \dotplus p_2)! \Rightarrow (p_1 \dotplus p_2) = (p_2 \dotplus p_1),\\
&((p_1 \dotplus p_2) \dotplus p_3)! \Rightarrow p_1 \dotplus (p_2 \dotplus p_3) = (p_1 \dotplus p_2) \dotplus p_3.
\end{aligned}
$$

Proof. Easy. By way of example we check $(p_1 + p_2)! \Rightarrow p_1 + p_2 = p_2 + p_1$. So assume that $(p_1 + p_2)!$ which means that their domains are disjoint. In order to show

[7] We have chosen the symbol $\dotplus$ because it is a combination of $+$ symbol and an $=$ symbol. Some care is needed not to confuse it with $\neq$, which means 'not equal', as usual.

[8] We write $p!$ if the expression p is defined. We adopt the conventions that '!' is a strict operator by which we mean that for every expression $p(q)$ in which q occurs as a subexpression, (not $q!$) implies (not $p(q)!$). Equality is strict too, by which we mean that $p_1 = p_2$ implies $p_1!$ and $p_2!$. As a consequence, $p! \Leftrightarrow (p = p)$. Readers familiar with COLD-K or COLD-1 recognize '!' as the definedness of these languages. These languages are based on the logic MPL_ω (see [9,10] or [13], respectively).

that $p_1 + p_2 = p_2 + p_1$ it suffices to show that $p_1 + p_2$ and $p_2 + p_1$ have the same domain (easy, the domain is the union of the domains of p_1 and p_2) and that for arbitrary a in the domain we find that $(p_1 + p_2)(a) = (p_2 + p_1)(a)$. There are two cases: if a is in the domain of p_1, then $(p_1 + p_2)(a) = p_1(a)$ and $(p_2 + p_1)(a) = p_1(a)$. Otherwise a is in the domain of p_2, and then $(p_1 + p_2)(a) = p_2(a)$ and $(p_2 + p_1)(a) = p_2(a)$. □

Another definition of the addition $+$, can be obtained by starting from an operator $+$ which works on singleton sets as follows: $\emptyset + \emptyset = \emptyset$, $\{a\} + \emptyset = \{a\}$, $\emptyset + \{b\} = \{b\}$ and $\{a\} + \{b\} =$ 'undefined'; the operator on functions follows by point-wise application.

2.6. Extending partial functions

Next we present another mathematical tool and its category-theoretic motivation. The tool is a way of extending a given denotation, say δ, to get a δ' which works on terms. Similarly we may want to extend typing τ to τ'.

Definition ($\text{NAM}^{(\,)}$)**.** Let NAM be a fixed, given set, whose elements are called 'names'. We define that $\text{NAM}^{(\,)}$ is the smallest set such that $\text{NAM} \subseteq \text{NAM}^{(\,)}$ and such that whenever $f \in \text{NAM}$ and $a \in \text{NAM}^{(\,)}$, we find that $f(a) \in \text{NAM}^{(\,)}$. The elements of $\text{NAM}^{(\,)}$ are called applicative expressions.

If we have a mapping $h : \text{NAM} \to B$ where B is some set equipped with an operator $\cdot((\cdot)) : B \times B \to B$, then we can write down the following recursion equations:

$$h'(n) = h(n),$$
$$h'(f(a)) = h(f)((h'(a))),$$

which defines a new mapping $h' : \text{NAM}^{(\,)} \to B$.

We assume that this works for partial functions too, say $h : \text{NAM} \rightharpoonup B$ or $\cdot((\cdot)) : B \times B \rightharpoonup B$.

2.7. Set-wise operators

Next we investigate how turning a binary operator into a set-wise binary operator is done. If we have a given binary operator, $+$ say, with $+ : A \times A \to A$ we can extend it to a binary operator in $\mathscr{P}(A) \times \mathscr{P}(A) \to \mathscr{P}(A)$. The definition of the new operator is for $s_1, s_2 \in \mathscr{P}(A)$

$$\langle s_1, s_2 \rangle \mapsto \{t_1 + t_2 \mid t_1 \in s_1 \wedge t_2 \in s_2\}.$$

Let us write $\mathscr{P}_{\square}(+)$ for the operator thus defined. It has to be distinguished from the usual functor $\mathscr{P}$ which is given by $\mathscr{P}(+) : \mathscr{P}(A \times A) \to \mathscr{P}(A)$. The correspondence

is given by the commuting diagram

$$\begin{array}{ccc} A \times A & \xrightarrow{+} & A \\ \downarrow_{\mathscr{P}} & & \downarrow_{\mathscr{P}} \\ \mathscr{P}(A \times A) & \xrightarrow{\mathscr{P}(+)} & \mathscr{P}(A) \\ \uparrow_{i} & \nearrow_{\mathscr{P}_{\square}(+)} & \\ \mathscr{P}(A) \times \mathscr{P}(A) & & \end{array}$$

where i maps $\langle s_1, s_2 \rangle$ to $\{\langle t_1, t_2 \rangle \mid t_1 \in s_1 \wedge t_2 \in s_2\}$. In a Cartesian plane, considering subsets of the plane, the elements of $\mathscr{P}(A \times A)$ are arbitrary subsets (e.g. blobs), whereas the elements of $\mathscr{P}(A) \times \mathscr{P}(A)$ are rectangles, as pointed out to us by Frank van der Linden. Then i maps each rectangle to the same rectangle, viewed as a blob. Although it is tempting to view i as an injection, formally it is not because all pairs containing $\emptyset$ are collapsed onto $\emptyset$. Sometimes we may just use the same notation for the lifted operators, omitting $\mathscr{P}$ and $\mathscr{P}_{\square}$.

3. Constructions for inheritance

Whereas the survey of our mathematical tools only mentioned abstract sets A, B and C, we now turn to three more specific sets, NAM, TYP and ELM. Of these NAM was inroduced in Section 2.6; its members are called *names*. The members of TYP and ELM are called *types* and *elements*, respectively. For these we shall define a number of useful constructions, amongst which several variants of the addition operators $\dashv\!\!\!+$, $+$ and $+\!\!\!\vdash$.

3.1. Assumptions

We assume primitive sets NAM, TYP and ELM. The members of ELM may be either values, variables or functions; the distinction is not made here. A *denotation* is a relation

$$\delta \subseteq \text{NAM} \times \text{ELM}.$$

A number of well-known issues are just algebraic properties, as we will illustrate next. For example, let us consider the issue of name clashes: it is customary to say that a name clash arises if the same name has been assigned twice, to distinct semantical elements. If all naming information is stored in a denotation δ (think of it as a symbol table) then we find that a name clash arises precisely if and only if δ is not functional. The opposite issue arises if two or more names associated with a single semantic element (just as the situation of one person who operates under two different names). In that case we find that δ is not injective. So δ has 'clashes' if it is not functional and 'aliases' if it is not injective. Similarly we say that δ has 'garbage' if it is not surjective, and 'dangling' references if it is not total.

Definition ($\text{TYP}^{\rightarrow}$)**.** The set $\text{TYP}^{\rightarrow}$ is defined as the smallest set which includes TYP and such that whenever t_1 and t_2 are in TYP we find that $t_1 \rightarrow t_2$ is in $\text{TYP}^{\rightarrow}$. We say that $\text{TYP}^{\rightarrow}$ is generated by TYP and a single application of $\cdot \rightarrow \cdot$.

The present definition of $\text{TYP}^{\rightarrow}$ corresponds to the flat kinds of types of typical imperative programming languages (functions of zero arguments and functions of two or more arguments are no problem, but here we treat just the case of functions of one argument). For some purposes we might prefer another, more general, definition, which would be that the set $\text{TYP}^{\rightarrow}$ is the smallest set which includes TYP and such that whenever t_1 and t_2 are in the set, so is $t_1 \rightarrow t_2$. In that case we would say that $\text{TYP}^{\rightarrow}$ is generated by TYP and $\cdot \rightarrow \cdot$.

Now a *typing* is a relation

$$\tau \subseteq \text{NAM} \times \text{TYP}^{\rightarrow},$$

which may or may not be functional.

Next we define the concept of overloading, which in a certain way, is a combination of a denotation and a typing. An *overloading* is a relation

$$\omega \subseteq \text{NAM} \times (\text{TYP}^{\rightarrow} \times \text{ELM}).$$

When viewed as a ternary relation, it may or may not be functional in its first argument (usually it is not), and similarly, it may or may not be functional in its first two arguments (usually it is, in which case we are able to study resolution).

We assume that the algebra of $\text{TYP}^{\rightarrow}$ is equipped with an applicative operator $\cdot((\cdot))$ on types, which is partial, and which is defined by

$$(t_1 \rightarrow t_2)((t_1)) = t_2.$$

We assume that the algebra of ELM is equipped with an applicative operator $\cdot(\cdot)$ on elements, which is partial. The intuition is that we may think of $f(e)$ for $f \in \text{ELM}$ and $e \in \text{ELM}$ as f being a function which is applied to e being an element in the domain of f, but since we do not investigate any deep semantical issues, we need not adopt any formal assumptions to enforce this intuition.

3.2. Adding typings

We have adopted the view that typings τ are subsets of a product of the form $A \times B$, taking NAM for A and $\text{TYP}^{\rightarrow}$ for B. Typings may or may not be functional.

If two typings are functional, the definitions of $\dashv\!\!\!+$, $+$ and $+\!\!\!\vdash$ are applicable.

Now the interesting thing is that, under certain assumptions, we may invent still other kinds of functionality and additions, exploiting the structure of $\text{TYP}^{\rightarrow}$. In particular, we shall propose a notion of functionality denoted as 'functional$_{1\frac{1}{2}}$' (adapting the notion with the same name introduced in Section 2.2). And for typings which are functional$_{1\frac{1}{2}}$ we propose special additions denoted as $\dashv\!\!\!+_{1\frac{1}{2}}$, $+_{1\frac{1}{2}}$ and $+\!\!\!\vdash_{1\frac{1}{2}}$.

We exploit the fact that $\text{TYP}^{\rightarrow}$ is the set such that $\text{TYP}^{\rightarrow} \supseteq \text{TYP}$ and such that whenever $t_1 \in \text{TYP}$ and $t_2 \in \text{TYP}$ we find that $(t_1 \rightarrow t_2) \in \text{TYP}^{\rightarrow}$. Therefore, we extend the set of allowed transformations (see Section 2.3) with two transformations (since there is no danger of confusion we refrain from subscripting $\cong$):

$$\begin{aligned} \text{TYP}^{\rightarrow} &\cong \text{TYP} + (\text{TYP} \times \text{TYP}) \\ \text{TYP} + (\text{TYP} \times \text{TYP}) &\cong (\mathbb{1} + \text{TYP}) \times \text{TYP}, \end{aligned}$$

where $\mathbb{1}$ is an arbitrary singleton set, which works as a neutral element with respect to Cartesian product formation and where $+$ denotes the so-called disjoint sum of two sets. More precisely $A+B$ is the set $\{\langle a,0\rangle \mid a \in A\} \cup \{\langle b,1\rangle \mid b \in B\}$; note that this gives rise to two injections $\text{in}_0 : A \rightarrow (A+B)$ and $\text{in}_1 : B \rightarrow (A+B)$ such that $\text{in}_0(a) = \langle a,0\rangle$ and $\text{in}_1(b) = \langle b,0\rangle$. We postulate that $\mathbb{1}$ is the set $\{o\}$ where o is an arbitrary object, not equal to some type in $\text{TYP}^{\rightarrow}$.

We say that a typing τ with $\tau \subseteq \text{NAM} \times \text{TYP}^{\rightarrow}$ is $\text{functional}_{1\frac{1}{2}}$ if for no pair $\langle n,t\rangle$ in τ with $t \in \text{TYP}$ we find another $t' \in \text{TYP}$ with $t' \neq t$ such that $\langle n,t'\rangle$ occurs in τ, and moreover, for no pair $\langle n,(t_1 \rightarrow t_2)\rangle$ in τ we find another t_2' with $t_2' \neq t_2$ such that $\langle n,(t_1 \rightarrow t_2')\rangle$ occurs in τ (here we adapt the notion of Section 2.2).[9] Therefore, it is possible to transform τ to $\tau_{1\frac{1}{2}} : \text{NAM} \times (\mathbb{1} + \text{TYP})) \supset\!\rightarrow \text{TYP}$ such that $\tau \cong \tau_{1\frac{1}{2}}$ (this will be demonstrated below) and then check whether $\tau_{1\frac{1}{2}}$ is functional in the ordinary sense.

Definition $(+_{1\frac{1}{2}})$**.** The extra transformations allow us to adopt the following construction as a definition for the addition $+_{1\frac{1}{2}}$ on typings. We shall use the obvious abbreviation to write A^2 for $A \times A$ for arbitrary set A. The $\downarrow_{\supseteq,\supseteq}$ step is critical; we return to it later.

$$\begin{array}{c} (\mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow}))^2 \\ \downarrow_{\cong} \\ (\mathscr{P}(\text{NAM} \times (\text{TYP} + (\text{TYP} \times \text{TYP}))))^2 \\ \downarrow_{\cong} \\ (\mathscr{P}(\text{NAM} \times ((\mathbb{1} + \text{TYP}) \times \text{TYP})))^2 \\ \downarrow_{\cong} \\ (\mathscr{P}((\text{NAM} \times (\mathbb{1} + \text{TYP})) \times \text{TYP}))^2 \\ \downarrow_{\supseteq,\supseteq} \end{array}$$

[9] If we would have $\tau \subseteq \text{NAM} \times (\text{TYP} \times \text{TYP})$ then the concept $\text{functional}_{1\frac{1}{2}}$ of Section 2.2 applies, saying that for no $\langle n,\langle t_1,t_2\rangle\rangle$ in τ we find $t_2' \neq t_2$ such that $\langle n,\langle t_1,t_2'\rangle\rangle \in \tau$. But instead of $\tau \subseteq \text{NAM} \times (\text{TYP} \times \text{TYP})$ we have to deal with $\tau \subseteq \text{NAM} \times \text{TYP}^{\rightarrow}$ with $\text{TYP}^{\rightarrow}$ not simply being $\text{TYP} \times \text{TYP}$, but the union of $\text{TYP} \times \text{TYP}$ and the unstructured set TYP. The adaptation is consistent with the definition of Section 2.2 because $\text{TYP}^{\rightarrow}$ can be transformed into a product, but we must make sure to adopt the proper transformation so that the extra TYP is treated as a result, not as an input.

$$((\textsc{nam} \times (\mathbb{1} + \textsc{typ})) \rightharpoondown \textsc{typ})^2$$
$$\downarrow_{+}$$
$$((\textsc{nam} \times (\mathbb{1} + \textsc{typ})) \rightharpoondown \textsc{typ})$$
$$\downarrow_{\subseteq}$$
$$\mathscr{P}((\textsc{nam} \times (\mathbb{1} + \textsc{typ})) \times \textsc{typ})$$
$$\downarrow_{\cong}$$
$$\mathscr{P}(\textsc{nam} \times ((\mathbb{1} + \textsc{typ}) \times \textsc{typ}))$$
$$\downarrow_{\cong}$$
$$\mathscr{P}(\textsc{nam} \times (\textsc{typ} + (\textsc{typ} \times \textsc{typ})))$$
$$\downarrow_{\cong}$$
$$\mathscr{P}(\textsc{nam} \times \textsc{typ}^{\rightarrow}).$$

Note that the operation thus defined is partial, in the sense that in order that $\tau_1 +_{1\frac{1}{2}} \tau_2$ is defined, we must demand that both τ_1 and τ_2 are functional$_{1\frac{1}{2}}$ (which comes on top of the fact that $+$ is a partial operation already).

This is the way to read the $\downarrow_{\supseteq,\supseteq}$ step in the above definition: as a way of telling that the operation defined by the whole sequence is partial. This should be contrasted to the later $\downarrow_{\subseteq}$ step which is concerned with transforming the result back to a larger domain, which will always succeed. In the $\textsc{nam} \times \textsc{typ}^{\rightarrow}$ view this means that the result of adding τ_1 and τ_2 by $+_{1\frac{1}{2}}$ contains all pairs of the form $\langle n, t\rangle$ derived from τ_1 as well as all pairs of the form $\langle n, (t_1 \to t_2)\rangle$ derived in τ_1 and all pairs $\langle n, t\rangle$ derived in τ_2 and all pairs $\langle n, (t_1 \to t_2)\rangle$ derived in τ_2. But there may not be a pair $\langle n, t\rangle$ in $\tau_1 \cap \tau_2$, nor may there be a pair $\langle n, (t_1 \to t_2)\rangle$ in τ_1 such that for some t_2' we find that $\langle n, (t_1 \to t_2')\rangle$ is in τ_2. In the analogous way we define $\nrightarrow_{1\frac{1}{2}}$ and $\mp_{1\frac{1}{2}}$ on typings.

For example, looking at $\nrightarrow_{1\frac{1}{2}}$ in the $\textsc{nam} \times \textsc{typ}^{\rightarrow}$ view, we see that the result of adding τ_1 and τ_2 contains all pairs of the form $\langle n, t\rangle$ derived from τ_1 as well as all pairs of the form $\langle n, (t_1 \to t_2)\rangle$ derived from τ_1 next to those pairs $\langle n, t\rangle$ derived from τ_2 for which there is no $\langle n, t'\rangle$ from τ_1, as well as those pairs $\langle n, (t_1 \to t_2)\rangle$ derived from τ_2 for which there is no $\langle n, (t_1 \to t_2')\rangle$ in τ_1.

3.3. Applications

From our earlier example we note that `Coffee` makes the following typing available for usage inside `Latte`. Let us call it τ_1.

Name	Type
b	int
r	int
ok	boolean → boolean

And inside `Latte` we find the definition of another typing. Let us call it τ_2.

Name	Type
m	int
ok	boolean → boolean
ok	int → boolean
test	int → boolean

Now `Latte` extends `Coffee`. The combined effect of all this is that inside `Latte` the total available set of variables and functions is given by $\tau_2 \not{+}_1 \tau_1$.

Name	Type
b	int
r	int
m	int
ok	boolean → boolean
ok	int → boolean
test	int → boolean

But note that it would be right according to the Java rules for `Latte` to redeclare the variable b as `boolean` or to define b once more to be of type `int`. It would be wrong however to redefine the function `ok` as a function of type `boolean` → `int`. The existing `ok` of type `boolean` → `boolean` can be redefined, but then the result type must be the same again, that is, `boolean`. This can be explained by saying that:

- the variables are added by $\not{+}_1$,
- the functions are added by $\ddagger_{1\frac{1}{2}}$,
- if one of these additions is undefined, the program is incorrect.

In practice, incorrect inheritance due to an addition which is undefined causes an error message to be produced by the compiler. The situation is also shown in Fig. 1.

We explain Fig. 1 now. The arrows depict the relation between language constructs where names are introduced and language constructs where names are used. For example the variable declaration section of `Coffee` introduces one or more names, which may be considered together as one typing. Similarly the variable declaration section of `Latte` introduces one or more names, which may be considered together as another typing. Variable names from both typings can be used in the method declaration section of `Coffee`, but for the method declaration their combined effect can be viewed as one single typing, viz. $\tau_1 +_1 \tau_2$. This is expressed by inserting the $\not{+}_1$ operator as a data-flow operator in the define-use arrows.

3.4. Adding overloadings

We adopt the view that overloadings ω_1, ω_2 are subsets of a product of the form $\mathrm{NAM} \times (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM})$. Overloadings can be functional in different senses: functional$_1$, functional$_{1\frac{1}{2}}$ or functional$_2$.

Definition ($+\!\!\!\!/_1$)**.** If two overloadings satisfy the strong requirement that, when considered as relations in one argument, they are functional, the definitions of $+$, $+\!\!\!\!/$ and $\ddagger$ are applicable. In that case we propose the following construction as a definition for the addition $+\!\!\!\!/$, also denoted as $+\!\!\!\!/_1$, on overloadings:

$$
\begin{array}{c}
(\mathscr{P}(\mathrm{NAM} \times (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM})))^2 \\
\downarrow_{\supseteq,\supseteq} \\
(\mathrm{NAM} \rightharpoondown (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM}))^2 \\
\downarrow_{+\!\!\!\!/} \\
\mathrm{NAM} \rightharpoondown (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM}) \\
\downarrow_{\subseteq} \\
\mathrm{NAM} \times (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM}).
\end{array}
$$

Note that the operation thus defined is partial, in the sense that in order that $\omega_1 +\!\!\!\!/_1 \omega_2$ is defined, we must demand that both ω_1 and ω_2 are functional$_1$ (that is, just functional).

Recall that overloadings are relations from names to pairs $\langle$type,element$\rangle$ and that we distinguished several (overlapping) classes of overloadings, viz. those which are functional$_1$, those which are functional$_{1\frac{1}{2}}$, and those which are functional$_2$. Those overloadings which are functional$_2$ can be converted ('rotated') to subsets of $(\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}) \times \mathrm{ELM}$ which (by definition of 'functional$_2$') are functional. Therefore our definitions of $+$ and $+\!\!\!\!/$ are applicable.

Definition ($+\!\!\!\!/_2$)**.** We propose the following construction as a definition for the addition $+\!\!\!\!/_2$ on overloadings:

$$
\begin{array}{c}
(\mathscr{P}(\mathrm{NAM} \times (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM})))^2 \\
\downarrow_{\cong} \\
(\mathscr{P}((\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}) \times \mathrm{ELM}))^2 \\
\downarrow_{\supseteq,\supseteq} \\
((\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}) \rightharpoondown \mathrm{ELM})^2 \\
\downarrow_{+\!\!\!\!/} \\
(\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}) \rightharpoondown \mathrm{ELM} \\
\downarrow_{\subseteq} \\
\mathscr{P}((\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}) \times \mathrm{ELM}) \\
\downarrow_{\cong} \\
\mathscr{P}(\mathrm{NAM} \times (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM})).
\end{array}
$$

Note that the operation thus defined is partial, in the sense that in order that $\omega_1 \dagger_2 \omega_2$ is defined, we must demand that both ω_1 and ω_2 are functional$_2$.

In this way we obtain the possibility to write just $\omega_1 \dagger_1 \omega_2$ and $\omega_1 \dagger_2 \omega_2$, without worrying much about the conversions from one representation to another.

Definition ($\dagger_{1\frac{1}{2}}$)**.** We adopt the following construction as a definition for the addition $\dagger_{1\frac{1}{2}}$ on overloadings:

$$
\begin{array}{c}
(\mathscr{P}(\mathrm{NAM} \times (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM})))^2 \\
\downarrow\cong \\
(\mathscr{P}(\mathrm{NAM} \times ((\mathrm{TYP} + (\mathrm{TYP} \times \mathrm{TYP})) \times \mathrm{ELM})))^2 \\
\downarrow\cong \\
(\mathscr{P}(\mathrm{NAM} \times (((\mathbb{1} + \mathrm{TYP}) \times \mathrm{TYP}) \times \mathrm{ELM})))^2 \\
\downarrow\cong \\
(\mathscr{P}((\mathrm{NAM} \times (\mathbb{1} + \mathrm{TYP})) \times (\mathrm{TYP} \times \mathrm{ELM})))^2 \\
\downarrow\supseteq,\supseteq \\
((\mathrm{NAM} \times (\mathbb{1} + \mathrm{TYP})) \rightharpoonup (\mathrm{TYP} \times \mathrm{ELM}))^2 \\
\downarrow\dagger \\
((\mathrm{NAM} \times (\mathbb{1} + \mathrm{TYP})) \rightharpoonup (\mathrm{TYP} \times \mathrm{ELM})) \\
\downarrow\subseteq \\
\mathscr{P}((\mathrm{NAM} \times (\mathbb{1} + \mathrm{TYP})) \times (\mathrm{TYP} \times \mathrm{ELM})) \\
\downarrow\cong \\
\mathscr{P}(\mathrm{NAM} \times (((\mathbb{1} + \mathrm{TYP}) \times \mathrm{TYP}) \times \mathrm{ELM})) \\
\downarrow\cong \\
\mathscr{P}(\mathrm{NAM} \times ((\mathrm{TYP} + (\mathrm{TYP} \times \mathrm{TYP})) \times \mathrm{ELM})) \\
\downarrow\cong \\
\mathscr{P}(\mathrm{NAM} \times (\mathrm{TYP}^{\rightarrow} \times \mathrm{ELM})).
\end{array}
$$

Note that the operation thus defined is partial, in the sense that in order that $\omega_1 \dagger_{1\frac{1}{2}} \omega_2$ is defined, we must demand that both ω_1 and ω_2 are functional$_{1\frac{1}{2}}$.

In the $(\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}) \rightharpoonup \mathrm{ELM}$ view, the definition of $\dagger_{1\frac{1}{2}}$ means that the result of adding ω_1 and ω_2 contains all maplets of the form $\langle n, t\rangle \mapsto e$ derived from ω_1 as well as all maplets of the form $\langle n, (t_1 \rightarrow t_2)\rangle \mapsto e$ derived from ω_1 next to those maplets $\langle n, t\rangle \mapsto e$ derived from ω_2 for which there is no $\langle n, t'\rangle \mapsto e'$ from ω_1, as well as those maplets $\langle n, (t_1 \rightarrow t_2)\rangle \mapsto e$ derived from ω_2 for which there is no $\langle n, (t_1 \rightarrow t_2')\rangle \mapsto e'$ in ω_1.

Definition $(+_1,\ +_{1\frac{1}{2}},\ +_2,\ \dot{+}_1,\ \dot{+}_{1\frac{1}{2}},\ \dot{+}_2)$**.** We define $+_1$ as $+$. We define $+_{1\frac{1}{2}}$ and $+_2$ as the operations obtained in the fashion analogous to $\dot{+}_{1\frac{1}{2}}$, $\dot{+}_2$, respectively. And the same for $\dot{+}_1$, $\dot{+}_{1\frac{1}{2}}$ and $\dot{+}_2$.

Proposition. *For functional$_i$ overloadings* ω, ω_1, *etc.*

$$\omega \dot{+}_i \emptyset = \omega,$$
$$\emptyset \dot{+}_i \omega = \omega,$$
$$\omega \dot{+}_i \omega = \omega,$$
$$(\omega_1 \dot{+}_i \omega_2) \dot{+}_i \omega_3 = \omega_1 \dot{+}_i (\omega_2 \dot{+}_i \omega_3),$$
$$(\omega_1 \dot{+}_i \omega_2) \dot{+}_i \omega_1 = \omega_1 \dot{+}_i \omega_2,$$

$$\omega +_i \emptyset = \omega,$$
$$\emptyset +_i \omega = \omega,$$
$$(\omega +_i \omega)! \Rightarrow \omega = \emptyset,$$
$$(\omega_1 +_i \omega_2)! \Rightarrow \omega_1 +_i \omega_2 = \omega_2 +_i \omega_1,$$
$$((\omega_1 +_i \omega_2) +_i \omega_3)! \Rightarrow (\omega_1 +_i \omega_2) +_i \omega_3 = \omega_1 +_i (\omega_2 +_i \omega_3),$$

$$\omega \dot{+}_i \emptyset = \omega,$$
$$\emptyset \dot{+}_i \omega = \omega,$$
$$(\omega \dot{+}_i \omega) = \omega,$$
$$(\omega_1 \dot{+}_i \omega_2)! \Rightarrow (\omega_1 \dot{+}_i \omega_2) = (\omega_2 \dot{+}_i \omega_1),$$
$$((\omega_1 \dot{+}_i \omega_2) \dot{+}_i \omega_3)! \Rightarrow \omega_1 \dot{+}_i (\omega_2 \dot{+}_i \omega_3) = (\omega_1 \dot{+}_i \omega_2) \dot{+}_i \omega_3,$$

where i can be $1,\ 1\frac{1}{2}$ *or* 2.

3.5. Applications

From our earlier example we note that `Coffee` makes the following overloading available for usage inside `Latte`. Let us call it ω_1.

Name	Type	Element
b	int	e_b
r	int	e_r
ok	boolean → boolean	f_{ok_1}

And inside `Latte` we find the definition of another overloading. Let us call it ω_2.

Name	Type	Element
m	int	e_m
ok	boolean → boolean	f_{ok_2}
ok	int → boolean	f_{ok_3}
test	int → boolean	f_{test}

The combined effect of this is that inside `Latte` the total available set of variables and functions is given by

Name	Type	Element
b	int	e_b
r	int	e_r
m	int	e_m
ok	boolean → boolean	f_{ok_2}
ok	int → boolean	f_{ok_3}
test	int → boolean	f_{test}

This can be explained by saying that:

- the variables are added by $\dashv\!\!\!\vdash_1$,
- the functions are added by $\dashv\!\!\!\vdash_{1\frac{1}{2}}$,
- the correctness or incorrectness of the program depends on the question whether the addition of the corresponding typings is defined, as explained in Section 3.3.

The corresponding typings can be obtained from the overloadings by a straightforward projection (omit the third column from each table). If we keep in mind that the correctness or incorrectness is determined by the corresponding typings, then we may equally well say that the variables are added by $\dashv\!\!\!\vdash_{1\frac{1}{2}}$, which makes no difference; similarly, the functions may be added by $\dashv\!\!\!\vdash_2$, no difference.

So in Java, the meaning of the `extends` construct is given by the $\dashv\!\!\!\vdash_1$ operation (for variables) and the $\dashv\!\!\!\vdash_{1\frac{1}{2}}$ operation (for functions) as defined above. We summarize the situation in Fig. 2.

3.6. Other Java rules

Let us look at the Java rule that determines how the typing corresponding to the declaration of another variable is combined with the typing of the earlier variables.

For example, consider the definition of `Latte` up to the point where the first function `ok` is declared (by the `extends` construct). So we have a typing, say τ_{2a} given by

Name	Type
b	int
r	int
m	int
ok	boolean → boolean

Now, the second function `ok` adds to this a singleton typing, say τ_{2b}

Name	Type
ok	int → boolean

The effect is that we get the typing

Name	Type
b	int
r	int
m	int
ok	boolean → boolean
ok	int → boolean

Now what are the operators that govern this addition process? It can be checked that for the typings

- the variables are added by $+_1$,
- the functions are added by $+_{1\frac{1}{2}}$,
- if one of these additions is undefined, the program is incorrect.

Note that we may equally well say that the variables are added by $+_{1\frac{1}{2}}$, which makes no difference. The situation is depicted in Fig. 3.

Precisely in the same way, for the corresponding overloadings

- the variables are added by $+_1$,
- the functions are added by $+_{1\frac{1}{2}}$,
- the correctness or incorrectness of the program depends on the question whether the addition of the corresponding typings is defined, as explained above.

Taking to account that correctness is determined by the corresponding typings, we may equally well say that the variables are added by $+_{1\frac{1}{2}}$ or $+_2$, which makes no difference. Similarly, the functions may be added by $+_2$, the only difference being that $+_2$ is too liberal with respect to its precondition. The same picture as Fig. 3 applies again.

Let us also look at the transformation that takes place when opening a new, inner scope with { and }. If we have for example inside a function (method) definition the following program fragment

```
int x;
int y;
e1
{  int x;                                   //wrong
   int z;
   e2
}
```

where e_1 and e_2 are arbitrary expressions, then for example the first set of declarations (`x` and `y`) yields an overloading, say ω_1, whereas the second set of declarations (`x` and `z`) adds an overloading, say ω_2. The effect is that expression e_1 is checked and interpreted with respect to ω_1 and that e_2 is checked with respect to

$$\omega_1 +_1 \omega_2$$

It could equally well be checked with respect to $\omega_1 +_{1\frac{1}{2}} \omega_2$, this does not make a difference since the language has no syntax for adding local functions (otherwise we would certainly have to use $\omega_1 +_{1\frac{1}{2}} \omega_2$, as is the case for the `extends` construct studied later). We can*not* say that e_2 is checked with respect to $\omega_1 +_2 \omega_2$ since Java does not allow re-using a variable name with a distinct type, for example redeclaring `long x` would be an error. The situation is depicted in Fig. 4.

Because $+_1$ is a partial operation, it is considered an error to re-declare `x` as in the above example. So $+_1$ does not 'override'. Also introducing e.g. `long x` is not correct when there is already a `int x` (this would demand working with $+_2$ instead of $+_1$).

For parameters of functions, the situation is as follows. Consider for example

```
void move(int x)
{  int x;
   e
}
```

and let the overloading built-up by the parameter list be called ω_1 and let the overloading built-up by the local declarations be called ω_2, then e is checked against $\omega_1 \Cup_1 \omega_2$, so the parameters override the variables declared inside the { and }. Again we may use $\omega_1 \Cup_{1\frac{1}{2}} \omega_2$ just as well since we assume that arrow-type parameters are not allowed anyhow (assumption made in Section 3.1 where we defined $\text{TYP}^{\rightarrow}$ and also discussed in one of the footnotes to Section 3.1). (see Fig. 5).

4. Constructions for polymorphism

We shall use the mathematical tools of Section 2, but again not dealing with sets A, B and C, but with the specific sets NAM, TYP and ELM of *names*, *types* and *elements*. For these we shall add some extra assumptions and then investigate overloading resolution.

4.1. Assumptions

As before, we consider $\text{NAM}^{()}$, the set of applicative expressions generated by NAM and $\cdot(\cdot)$, by which we mean that $\text{NAM}^{()}$ is the smallest set which includes NAM and such that whenever $f \in \text{NAM}$ and $a \in \text{NAM}^{()}$, we find that $f(a) \in \text{NAM}^{()}$.

Let us assume that the set ELM is equipped with an operation, also denoted as $\cdot(\cdot)$, which we call (semantical) application.

4.2. Extending denotations

Now a functional denotation $\delta \subseteq \text{NAM} \times \text{ELM}$ is extended to $\delta' : \text{NAM}^{()} \rightharpoonup \text{ELM}$ in a homomorphic way. The extension can be described by the recursion equations from the following definition.

Note on notation: in this section we are slightly sloppy with respect to the formal difference between $\delta \in \mathscr{P}(\text{NAM} \times \text{ELM})$ and $\delta : \text{NAM} \rightharpoonup \text{ELM}$. When treating the extension of typings in a next section we shall be more formal with respect to the 'rotations' involved. **Definition** (*Extension* $'$)**.** The extension operator $'$ which maps a functional denotation $\delta : \text{NAM} \rightharpoonup \text{ELM}$ to $\delta' : \text{NAM}^{()} \rightharpoonup \text{ELM}$ is given by the following recursion equations

$$\delta'(n) = \delta(n)$$
$$\delta'(f(a)) = \delta(f)(\delta'(a))$$

for $n, f \in \text{NAM}$ and $a \in \text{NAM}^{()}$, where we refer to the explanations of Section 2.6.

This only works if the elements assigned by δ to f happens to be a function and the element assigned to argument a is a value (but these are semantic constraints which are outside the scope of our model, where we assume nothing more than an applicative operator $\cdot(\cdot)$ on ELM).

Example. Consider the Java functions

```
int inc(int i) {
    return (i + 1);
}

int dbl(int i) {
    return (i + i);
}
```

Then, this defines a denotation δ, which, using lambda-notation [4], is given as

Name	Element
`0`	0
`inc`	$(\lambda x.x + 1)$
`dbl`	$(\lambda x.x + x)$

Therefore, δ' is an infinite mapping, viz.

Name	Element
`0`	0
`inc(0)`	$(\lambda x.x + 1)(0) = 1$
`dbl(0)`	$(\lambda x.x + x)(0) = 0$
`inc(inc(0))`	$(\lambda x.x + 1)((\lambda x.x + 1)(0)) = 2$
`inc(dbl(0))`	$(\lambda x.x + 1)((\lambda x.x + x)(0)) = 1$
`dbl(inc(0))`	etc.

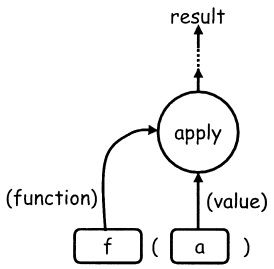

Fig. 6. Extension and denotations (δ').

The extension procedure is sketched graphically in Fig. 6. The idea sketched here is that `f` and `a` are syntactic elements (and so are the brackets). These syntactic elements are assigned by δ to a function (arrow labeled '`(function)`') and a value (arrow labeled '`(value)`'), respectively. The latter two semantic elements are combined by the `apply` operator, which depicts the applicative operator $\cdot(\cdot)$ on ELM. The dots towards the result convey the idea that this procedure is applied recursively.

4.3. *Extending typings* (1)

Now we want to do something similar to the extension process described in Section 4.2, but now for types, so let us have a look at types next. For example, the set TYP of basic types could include `long` and `String`. Recall that we restrict ourselves to simple types, excluding higher order types like $(t_1 \to t_2) \to t_3$ and excluding types with arrow-type results like $t_1 \to (t_2 \to t_3)$. Types for functions of two or more arguments are no problem but here we explain things for functions of one argument.

Recall the assumed operator $\cdot((\cdot))$ on types, which is partial, and which is defined by $(t_1 \to t_2)((t_1)) = t_2$. Recall that a typing τ is a subset of $\mathrm{NAM} \times \mathrm{TYP}^{\to}$. If a typing τ happens to be functional, we can use the canonical transformations of Section 2.3 and rotate the typing in another position, choosing $\tau_{(1)} \sim \tau$ such that we have a functional mapping $\tau_{(1)} : \mathrm{NAM} \rightharpoonup \mathrm{TYP}^{\to}$. In a diagram

$$\begin{array}{ccc} \tau & \in & \mathscr{P}(\mathrm{NAM} \times \mathrm{TYP}^{\to}) \\ \downarrow & & \downarrow_{\supseteq} \\ \tau_{(1)} & \in & \mathrm{NAM} \rightharpoonup \mathrm{TYP}^{\to}. \end{array}$$

The latter $\tau_{(1)}$ is extended to $\tau'_{(1)} : \mathrm{NAM}^{(\,)} \rightharpoonup \mathrm{TYP}^{\to}$ as described by the recursion equations

$$\begin{aligned} \tau'_{(1)}(n) &= \tau_{(1)}(n), \\ \tau'_{(1)}(f(a)) &= \tau_{(1)}(f)((\tau'_{(1)}(a))). \end{aligned}$$

So $\tau_{(1)}$ is mapped to $\tau'_{(1)}$ in a homomorphic way, but in the type-space the application operator, for fixed second argument, is the inverse of arrow abstraction, for fixed first

argument. This is an algebraic formulation of the insight that application and abstraction are dual things: formally $\cdot(\!(\, t_1 \,)\!)$ and $t_1 \rightarrow \cdot$ are each others inverses.

Once the extension has been done, we can rotate back the result into the original position.

Definition (*Extension* $'$)**.** Formally, the construction of the extension operator $'$ from $\mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow})$ to $\mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow})$ is given by the construction below, using the extension operator $'$ from $\text{NAM} \rightharpoonup \text{TYP}^{\rightarrow}$ to $\text{NAM}^{()} \rightharpoonup \text{TYP}^{\rightarrow}$ which maps $\tau_{(1)}$ to $\tau'_{(1)}$ as defined by the above-mentioned recursion equations

$$
\begin{array}{c}
\mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow}) \\
\downarrow_{\supseteq} \\
\text{NAM} \rightharpoonup \text{TYP}^{\rightarrow} \\
\downarrow_{\text{recursion}\ '} \\
\text{NAM}^{()} \rightharpoonup \text{TYP}^{\rightarrow} \\
\downarrow_{\subseteq} \\
\mathscr{P}(\text{NAM}^{()} \rightharpoonup \text{TYP}^{\rightarrow}).
\end{array}
$$

Note the first $\supseteq$ step which says that the extension operator $'$ thus defined only works for functional typings. Now τ' yields basic types only.

Since τ assigns basic types to constants and arrow-types to functions, we find that the well-typed applicative expressions are those getting a basic type again via τ'.

Example. In this example we illustrate the working of $'$. Suppose TYP has elements nat, int, long and float. Therefore $\text{TYP}^{\rightarrow}$ has, amongst others, elements $(\text{int} \rightarrow \text{int})$ and $(\text{nat} \rightarrow \text{nat})$. Let τ be given by the following table

Name	Type
i	int
nxt	int → int

Then, we find that $\tau'(\text{nxt(i)}) = (\text{int} \rightarrow \text{int})(\!(\text{int})\!) = \text{int}$, as expected. But if we would have defined τ as

Name	Type
i	int
nxt	int → int
nxt	nat → nat

then, we find that τ' is not defined since τ fails to be functional. Therefore we cannot evaluate $\tau'(\text{nxt(i)})$.

4.4. Extending typings (2)

There are several ways of doing resolution, by which we mean finding out the unique type of an expression when one or more of the subterms are not uniquely typed. Each such way gives rise to an extension of τ, notably τ', τ'' (see below), τ''' (further below), τ'''' (like τ''' but meant for subtyping). In each of these cases, our task is to depart from a given typing $\tau \subseteq \mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}$ and then construct an element of $\mathrm{NAM}^{()} \rightharpoonup \mathrm{TYP}^{\rightarrow}$. The simplest construction is the extension of τ to τ' already treated, using a simple recursion; but this only works under the severe restriction that τ is functional.

Any typing τ, even if it is not functional, can be rotated in another position, choosing $\tau_{(2)} \sim \tau$ such that we have a functional mapping $\tau_{(2)} : \mathrm{NAM} \to \mathscr{P}(\mathrm{TYP}^{\rightarrow})$. In a diagram

$$\begin{array}{ccc} \tau & \in & \mathscr{P}(\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}) \\ \downarrow & & \downarrow\cong \\ \tau_{(2)} & \in & \mathrm{NAM} \to \mathscr{P}(\mathrm{TYP}^{\rightarrow}). \end{array}$$

The latter $\tau_{(2)}$ is extended to $\tau''_{(2)} : \mathrm{NAM}^{()} \rightharpoonup \mathscr{P}(\mathrm{TYP}^{\rightarrow})$ as described by the recursion equations

$$\begin{aligned} \tau''_{(2)}(n) &= \tau_{(2)}(n) \\ \tau''_{(2)}(f(a)) &= \tau_{(2)}(f)((\tau''_{(1)}(a)))_{\mathscr{P}\square} \end{aligned}$$

where $\cdot((\cdot))_{\mathcal{P}\square}$ is the set-wise version of $\cdot((\cdot))$ (as discussed in Section 2.7 where it was proposed to write $\mathscr{P}_{\square}(+)$ for the set-wise version of any '+' operation). The formal definition is that $s_1((s_2))_{\mathscr{P}\square} = \{t_1((t_2)) \mid t_1 \in s_1, t_2 \in s_2\}$ for all sets s_1 and s_2.

So $\tau_{(2)}$ is mapped to $\tau''_{(2)}$ in a homomorphic way, again, but now using the powerset algebra of types. Once the extension has been done, we can rotate back the result to the original position, provided an additional check for uniqueness is added.

Definition (*Extension* $''$)**.** Formally, the construction of the extension operator $''$ from $\mathscr{P}(\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow})$ to $\mathscr{P}(\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow})$ is given by the construction below, using the extension operator $''$ from $\mathrm{NAM} \rightharpoonup \mathscr{P}(\mathrm{TYP}^{\rightarrow})$ to $\mathrm{NAM}^{()} \rightharpoonup \mathscr{P}(\mathrm{TYP}^{\rightarrow})$ which maps $\tau_{(2)}$ to $\tau''_{(2)}$ as defined by the above-mentioned recursion equations

$$\begin{array}{c} \mathscr{P}(\mathrm{NAM} \times \mathrm{TYP}^{\rightarrow}) \\ \downarrow\cong \\ \mathrm{NAM} \to \mathscr{P}(\mathrm{TYP}^{\rightarrow}) \\ \downarrow_{\text{recursion }''} \\ \mathrm{NAM}^{()} \rightharpoonup \mathscr{P}(\mathrm{TYP}^{\rightarrow}) \\ \downarrow\forall\mathscr{U} \end{array}$$

Table 4

Name	Type
i	int
i	nat
n	nat
nxt	int → int
nxt	nat → nat
lft	nat → float

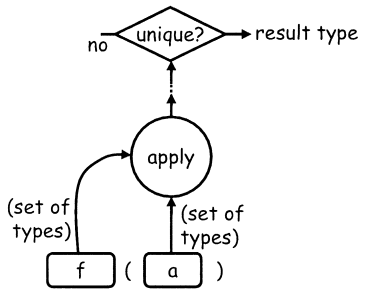

Fig. 7. Extension and typings (τ'').

$$\begin{array}{c} \mathrm{NAM}^{(\,)} \to \mathscr{P}_{\leqslant 1}(\mathrm{TYP}^{\to}) \\ \downarrow\cong \\ \mathrm{NAM}^{(\,)} \rightharpoonup \mathrm{TYP}^{\to} \\ \downarrow\subseteq \\ \mathscr{P}(\mathrm{NAM}^{(\,)} \times \mathrm{TYP}^{\to}). \end{array}$$

It can be checked that τ'' yields basic types only.

The demand for uniqueness with the aim of arriving at singleton sets at some point is the essence of overloading resolution.

Example. In this example we illustrate the working of $''$. Let τ be given by Table 4. Then, we find that $\tau''(\mathrm{nxt(n)})$ equals the unique element in $\{(\mathrm{int}\to\mathrm{int}),(\mathrm{nat}\to\mathrm{nat})\}((\{\mathrm{nat}\}))_{\mathscr{P}_\square} = \{\mathrm{nat}\}$, i.e. nat. We can also evaluate $\tau''(\mathrm{lft(nxt(i))})$ which equals the unique element in $\{(\mathrm{nat}\to\mathrm{float})\}((\{(\mathrm{int}\to\mathrm{int}),(\mathrm{nat}\to\mathrm{nat})\}((\{\mathrm{int},\mathrm{nat}\}))_{\mathscr{P}_\square}))_{\mathscr{P}_\square} = \{(\mathrm{nat}\to\mathrm{float})\}((\{\mathrm{int},\mathrm{nat}\}))_{\mathscr{P}_\square} = \{\mathrm{float}\}$, i.e. float. But there still exist terms for which τ'' yields undefined, for example $\tau''(\mathrm{nxt(i)})$ is undefined.

The extension procedure is sketched graphically in Fig. 7.

4.5. *Extending typings* ($1\frac{1}{2}$)

Our next idea is to rotate a typing τ which is functional$_{1\frac{1}{2}}$ into the corresponding position of $\tau_{(1\frac{1}{2})}$ and then, subtly reformulating the recursion equations, extend this $\tau_{(1\frac{1}{2})}$ so as to get an extension $\tau'''_{(1\frac{1}{2})}$ which works on $\mathrm{NAM}^{(\,)}$.

If τ is functional$_{1\frac{1}{2}}$ we can rotate it to get $\tau_{(1\frac{1}{2})} \sim \tau$ where $\tau_{(1\frac{1}{2})} : \text{NAM} \times (\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP}$. By further Currying we obtain $\tau_{(1\frac{1}{2}c)}$ as follows:

$$\begin{array}{ccc}
\tau & \in & \mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow}) \\
\downarrow & & \downarrow \cong \supseteq \\
\tau_{(1\frac{1}{2})} & \in & \text{NAM} \times (\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP} \\
\downarrow & & \downarrow \cong \\
\tau_{(1\frac{1}{2}c)} & \in & \text{NAM} \to ((\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP}).
\end{array}$$

Note that sometimes we are making shortcuts, such as the arrow labeled $\cong \supseteq$ which indicates that first we make a $\cong$ step, followed by a $\supseteq$ step (this is to be distinguished from the pairwise steps which we applied to squared sets earlier).

The latter $\tau_{(1\frac{1}{2}c)}$ is extended to $\tau'''_{(1\frac{1}{2}c)} : \text{NAM}^{()} \to ((\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP})$ as described by the recursion equations

$$\begin{aligned}
\tau'''_{(1\frac{1}{2}c)}(n) &= \tau_{(1\frac{1}{2}c)}(n), \\
\tau'''_{(1\frac{1}{2}c)}(f(a)) &= \lambda x.\tau_{(1\frac{1}{2}c)}(f)(\tau'''_{(1\frac{1}{2}c)}(a)(x)).
\end{aligned}$$

So $\tau_{(1\frac{1}{2}c)}$ is mapped to $\tau'''_{(1\frac{1}{2}c)}$ in a homomorphic way, not mapping to the powerset algebra, but to an algebra of functions whose operator is just function composition. Note that we may rewrite the last equation as $\tau'''_{(1\frac{1}{2}c)}(f(a)) = \tau_{(1\frac{1}{2}c)}(f) \circ \tau'''_{(1\frac{1}{2}c)}(a)$. Once the extension has been done, we can rotate back the result to the original position.

Definition (*Extension* $'''$)**.** Formally, the construction of the extension operator $'''$ from $\mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow})$ to $\mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow})$ is given by the construction below, using the extension operator $'''$ from $\text{NAM} \to ((\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP})$ to $\text{NAM}^{()} \to ((\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP})$ which maps $\tau_{(1\frac{1}{2}c)}$ to $\tau'''_{(1\frac{1}{2}c)}$ as defined by the above-mentioned recursion equations

$$\begin{array}{c}
\mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow}) \\
\downarrow \cong \\
\mathscr{P}(\text{NAM} \times ((\mathbb{1} + \text{TYP}) \times \text{TYP})) \\
\downarrow \cong \\
\mathscr{P}((\text{NAM} \times (\mathbb{1} + \text{TYP})) \times \text{TYP}) \\
\downarrow \supseteq \\
\text{NAM} \times (\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP} \\
\downarrow \cong \\
\text{NAM} \to ((\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP}) \\
\downarrow \text{recursion } ''' \\
\text{NAM}^{()} \to ((\mathbb{1} + \text{TYP}) \rightharpoonup \text{TYP}) \\
\downarrow \cong \\
(\text{NAM}^{()} \times (\mathbb{1} + \text{TYP})) \rightharpoonup \text{TYP} \\
\downarrow \text{take } o \text{ for 2nd arg}
\end{array}$$

$$\begin{array}{c} \mathrm{NAM}^{()} \rightharpoonup \mathrm{TYP} \\ \downarrow\subseteq \\ \mathscr{P}(\mathrm{NAM}^{()} \times \mathrm{TYP}) \\ \downarrow\subseteq \\ \mathscr{P}(\mathrm{NAM}^{()} \times \mathrm{TYP}^{\rightarrow}). \end{array}$$

It can be checked that τ''' yields basic types only.

Example. In this example we illustrate the working of $'''$. Let τ be given by Table 5. We find that $\tau_{(1\frac{1}{2}c)}$ is

Name	Type transformer
n	type → type: o → nat
nxt	type → type: int → int; nat → nat
lft	type → type: nat → float

and therefore we find that $\tau'''_{(1\frac{1}{2}c)}$ equals

Name	Type transformer
n	as before
nxt	as before
lft	as before
nxt(n)	type → type: o → nat
nxt(nxt)	type → type: int → int; nat → nat
	etc.

Therefore, $\tau'''(\mathrm{nxt}(\mathrm{n})) = \tau'''_{(1\frac{1}{2}c)}(\mathrm{nxt}(\mathrm{n}))(o) = \mathrm{nat}$.

Table 5

Name	Type
n	nat
nxt	int → int
nxt	nat → nat
lft	nat → float

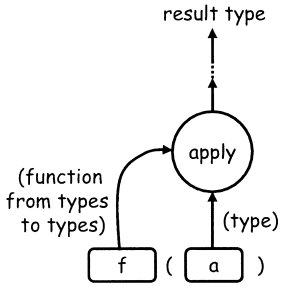

Fig. 8. Extension and typings (τ''').

But if we would overload 'i', by adopting τ to be

Name	Type
i	int
i	nat
n	nat
nxt	int → int
nxt	nat → nat
lft	nat → float

then we find that we cannot evaluate $\tau'''(\mathrm{lft}(\mathrm{nxt}(\mathrm{i})))$ because τ is not functional$_{1\frac{1}{2}}$.

The extension procedure is sketched graphically in Fig. 8 (the figure is a simplification in the sense that the step of taking o for the second argument is not visualized).

4.6. Discussion

In the table below we summarize what has been achieved until now. All three versions of extending typings are shown to be instances of one and the same idea. The idea is to map the applicative operator that generates NAM$()$ in a homomorphic way to

an operator in another algebra, the latter algebra being concerned with types.

Type algebra	Rotation	Effect
$\cdot(\!(\cdot)\!)$	$\tau_{(1)}$	Pascal style
$\cdot(\!(\cdot)\!)_{\mathscr{P}_\square}$	$\tau_{(2)}$	COLD [10] style
$\cdot \circ \cdot$	$\tau_{(1\frac{1}{2}c)}$	Java style

We worked with three different operators to form such an algebra; the first operator is $\cdot(\!(\cdot)\!)$, defined by the rule that $\cdot(\!(t)\!)$ inverses $t \to \cdot$. The second operator is the powerset version of the first operator. The third operator is nothing but function composition $\circ$. Depending on the chosen algebra, we need a different rotation to turn a given τ into a mapping whose range fits the desired operator. These rotations are $\tau_{(1)}$, $\tau_{(2)}$ and $\tau_{(1\frac{1}{2}c)}$, respectively.

The approach can be extended in several ways. One topic is how typing steers denotation, that is, how to extend a given overloading ω to find ω', ω'' or ω''', analogously to τ', τ'' or τ''' respectively; this is fairly straightforward. Another topic is subtyping: there exists an extension procedure $''''$ which generalizes $'''$ but which can handle subtypes as well (such analysis of resolution is based on the assumption that types are always determined at compile time, whereas Java uses dynamic information; fortunately the object's type is always a subtype of the type calculated by a τ'''' construction and since an object of a subtype has always *more* fields and methods, it is safe to typecheck using τ''''). Because of space limitations we shall not address these topics here.

5. Constructions for encapsulation

The efforts of this section are aimed at a formalization of the following phenomenon: accessing a variable field v of a class c is to be done in two different ways, depending on the relative placement of the defining and the applied occurrence of v. Inside the class definition of c, it is possible to access the field v by just writing v, which is interpreted as the v of the object 'this'. But outside the scope of the defining class, it is necessary to write $e.v$ where e is an expression of type c. Similarly, invoking function f of class c is written as $f(i)$, but outside the scope of the defining class, it is necessary to write $e.f(i)$ where e is an expression of type c. Inside a class d which inherits from c, just $f(i)$ is correct again, however.

We show a slightly modified version of the example of Section 1.2.[10]

```
class Wine {
   public int a; // alcohol
   public int s; // sulfite
```

[10] In order to show that we are dealing with a phenomenon which exists independently of overloading and inheritance, we have chosen distinct names for the various `ok` functions, and we have added test functions to `Wine` and `Coffee` (similar to the earlier `compare` of `Latte`).

```
    public int wine_ok(int i) {
        return abs(a - i) - 2*s;
    }

    int wine_test(Wine w) {
    {   int x;
        x = wine_ok(12) - w.wine_ok(12);
        return x;
} }
class Coffee {
    public int b; // blackness
    public int r; // residuals

    public int coffee_ok(int i) {
        return abs(b - i) - r;
} }
```

We can see that inside the class `Wine` it is possible to write an invocation as `wine_ok(12)` indeed, whereas outside the scope of the defining class `Wine` it is necessary to write `w.wine_ok(12)`, `w` being an expression of type `Wine` (12 is the ideal value for good wine's alcohol percentage).

5.1. Assumptions

It is possible to view a method such as `wine_ok(int i)` as a function of two arguments, one argument having type `Wine`, the other argument having type `int`. Yet it is wrong to denote its application as `wine_ok(w,12)`, or `12.w.wine_ok()`, and we note that this difference is related to the presence of two distinct applicative operators, the dot and the parentheses. In order to make the typing system aware of this distinction, we adopt two distinct type abstraction operators, denoted as $\rightarrow_.$ and $\rightarrow$ respectively (the only notational difference is the small dot). In this approach, considering an applied occurrence of `wine_ok` in the scope of `Coffee`, we find that `wine_ok` has type $\texttt{Wine} \rightarrow_. (\texttt{int} \rightarrow \texttt{int})$.

We define $\text{TYP}^{\rightarrow_.,\rightarrow}$ as the smallest set which includes TYP and which is closed under at most one application of $\rightarrow$ and at most one outermost application of $\rightarrow_.$. So for types t_1, t_2 and $t_3 \in \text{TYP}$ we get types like t_1, $t_2 \rightarrow t_1$, $t_3 \rightarrow_. t_1$, and $t_3 \rightarrow_. (t_2 \rightarrow t_1) \in \text{TYP}^{\rightarrow_.,\rightarrow}$.

We stipulate that there is an applicative operator $\cdot((\cdot))$ which is the inverse of the type abstraction $\rightarrow$ as before. Since we have a new kind of type abstraction $\rightarrow_.$ we propose a new kind of type application, which is denoted by $\bullet$ and whose defining equation is

$$t_1 \bullet (t_1 \rightarrow_. t_2) = t_2$$

so $\bullet$ resembles $\cdot((\cdot))$ but has reversed function-argument order (just like dot application).

Table 6

Name	Type
a	int
s	int
wine_ok	int → int

Table 7

Name	Type
a	Wine →. int
s	Wine →. int
wine_ok	Wine →. (int → int)

We assume that dot application takes priority over the application indicated by the parentheses and that for example `wine_ok` (in the scope of `Coffee`) has type `Wine` $\rightarrow_.$ `(int` $\rightarrow$ `int)`. This has the effect that `w.wine_ok(12)` is well-typed, since `w.wine_ok` has type `int` $\rightarrow$ `int`.

5.2. Encapsulating typings

We begin with an example. Let τ_{Wine} be given by Table 6 which is a set of variables and functions as declared and made available inside and outside `Wine` of the previous section.[11] When the functions of this typing τ_{Wine} are used outside `Wine`, the typing has become different, namely as shown in the Table 7.

We want to view the typing of the latter table as the result of applying an abstraction operator, also denoted as $\rightarrow_.$ working on the entire table. In other words, we propose a lifted version of $\rightarrow_.$ which works at the level of typings, not individual names. Since we define this abstraction operator, we adopt the corresponding application operator as well.

First, we define the (infix) operator $\rightarrow_. : \text{TYP} \times \text{TPN} \rightarrow \text{TPN}$. Here $\text{TPN} = \mathscr{P}(\text{NAM} \times \text{TYP}^{\rightarrow_.\rightarrow})$, that is TPN is the set of typings (a typing being a relation from names to types, not necessarily a functional relation). In its intended usage $c \rightarrow_. \tau$, the type c is the name of a class.

Definition ($c \rightarrow_. \cdot$). We define the result of $c \rightarrow_. \tau$ by the prescription: replace each name-type pair $n \mapsto t$ occurring in τ by $n \mapsto (c \rightarrow_. t)$, for the given fixed c. Formally $c \rightarrow_. \tau = \{n \mapsto (c \rightarrow_. t) \mid (n \mapsto t) \in \tau\}$. Note that this is nothing but just an application of our $\forall$ functor.

[11] We could say that certain invocations of `wine_ok` have to be repaired because somehow it was allowed to forget the prefix '`this.`' and that a preprocessing phase adds such occurrences of '`this.`'. But we do not want to adopt this viewpoint and instead we want to have operators for all phenomena rather than a preprocessor.

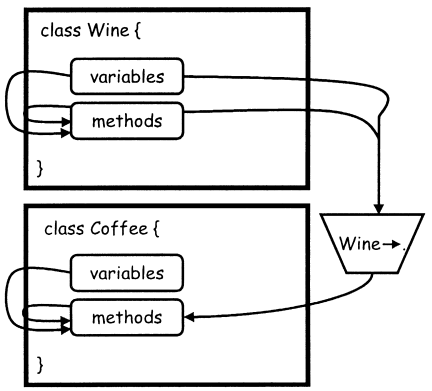

Fig. 9. Encapsulation and typings.

The next operator is $\bullet : \text{TYP} \times \text{TPN} \to \text{TPN}$. The effect of $c \bullet \cdot$ is inverse to that of $c \to_. \cdot$ where again, when using $c \bullet \tau$, the name c is the name of a class.

Definition $(c \bullet \cdot)$. We define the result of $c \bullet \tau$ by the following prescription: replace each maplet $n \mapsto (c \to_. t)$ occurring in τ by $n \mapsto t$, and remove all pairs which are not of the form $n \mapsto (c \to_. t)$. Formally $c \bullet \tau = \{n \mapsto t \mid n \mapsto (c \to_. t) \in \tau\}$.

The following laws hold for all $\tau \in \text{TPN}$ and for functional $\tau_1,\ \tau_2 \in \text{TPN}$, provided we consider only $c,\ \tau,\ \tau_1,\ \tau_2$ for which both sides of the equations are defined.

$$
\begin{aligned}
c \bullet \emptyset &= \emptyset,\\
c \to_. \emptyset &= \emptyset,\\
c \to_. (c \bullet \tau) &\subseteq \tau,\\
c \bullet (c \to_. \tau) &= \tau,\\
c \bullet (\tau_1 \dagger \tau_2) &= (c \bullet \tau_1) \dagger (c \bullet \tau_2),\\
c \to_. (\tau_1 \dagger \tau_2) &= (c \to_. \tau_1) \dagger (c \to_. \tau_2).
\end{aligned}
$$

We have analogous distribution laws for $+$ and $\dotplus$.

As explained already for the `Wine` example, the situation is as shown in Fig. 9.

So we were able to model the transformation that applies to a typing when leaving the scope of a class. But what happens to the corresponding denotation? And what happens to the corresponding overloading? Are we able to model transformations that apply to meanings and overloadings when leaving the scope of a class? We found that the approach can be extended to answer the latter question positively, but in view of space limitations we only sketch the basic idea: we assume that there is an operator, denoted by the *hat* symbol, working on ELM. If e is a variable's address (an offset in an address space), then $\hat{e}$ is a function which transforms an object's address to a field address by adding the offset. We demand that there is a semantical

counterpart of the type-level application $\bullet$ which is denoted as an infix operator '.'. The situation is analogous to the type-level application $\cdot((\cdot))$ whose semantical counterpart is application of functions, denoted as $\cdot(\cdot)$. We postulate a neutral element to undo the effect of the hat action, so we assume an element $0 \in \text{ELM}$. We also need something to interpret the syntactic $\texttt{this} \in \text{NAM}$ for which we assume a semantic counterpart 'this' $\in \text{ELM}$. Using such assumptions, it is possible lift the hat function to denotations and develop operators like $e \bullet \delta$, $c \bullet_0 \omega$ and $c \rightarrow_. \omega$.

6. Concluding remarks

Achievements: a number of operators for manipulating relations and mappings from names to types and semantical elements have been defined and their properties investigated. The operators turn out applicable to the three object oriented themes: inheritance, polymorphism and encapsulation. Although not all peculiarities of Java have been captured we believe that our understanding of a number of useful and fundamental mechanisms for naming has improved.

We did not just produce a zoo of operations but we developed a set of relevant operations which follow from a few principles of great elegance and simplicity. These principles are the following:

- there are three ways of adding partial functions, one way modelled by a total operation $\dot{+}$, the other two ways unavoidably by partial operations, viz. $\dot{-}$ and $+$.
- there is a simple mechanism of extending a given mapping h whose domain is NAM to get a mapping h' whose domain is $\text{NAM}^{()}$. The mechanism is taking h' to be a unique homomorphism. In each case we start from a given operator which acts as the homomorphic image of the applicative operator $\cdot(\cdot)$ generating $\text{NAM}^{()}$. We worked with three examples of such operators. These are firstly, $\cdot((\cdot))$ obtained as an inverse of the type constructor $\rightarrow$, secondly the powerset version of the first operator. Surprisingly the third operator is nothing but function composition $\circ$.
- denotations, typings and overloadings are relations, so in order to apply an addition operation or an extension operation, they have to be transformed into (partial) functions, and once the addition or extension is done, the result is to be transformed back. We proposed a collection of such transformations, most of them being well-known mathematical ideas such as the rebracketing of products and Currying.

The operations $+_1$, $+_{1\frac{1}{2}}$, $+_2$, $\dot{+}_1$, $\dot{+}_{1\frac{1}{2}}$, $\dot{+}_2$, $\dot{-}$, $\dot{-}_{1\frac{1}{2}}$, $\dot{-}_2$, $'$, $''$ and $'''$ were obtained using hardly anything else than these principles.

We developed a constructive style to define complex operations, avoiding clumsy conversion functions being explictly mentioned, still being completely formal. We used notational conventions exploiting the above-mentioned transformations.

Using these operators, we were able to characterize a number of Java scope rules. We claim that Figs. 1–5, can be viewed as accurate and compact expressions of a number of relevant scope rules. Not only does this show how things work in Java, but

it also shows how the language could have been designed differently. Java's language designer could have chosen to use $+_1$ in Fig. 5 instead of $\dashv_1$.

Under the simplification of not treating subtypes, the extension operator $'''$ characterizes the Java way of doing resolution. Again, this is a designer's choice; as shown by the table in Section 4.6 Pascal's designer chose $'$ and COLD's designer chose $''$. The algebraic properties of the operators are helpful when judging the consistency of various choices. For example, the operations subscripted with i ($i = 1,\ 1\frac{1}{2},\ 2$) preserve the property of functionality$_j$ for $j \geqslant i$. Java makes sure that all the typings which assign types to method names are functional$_{1\frac{1}{2}}$. That is why we find no operators with subscript 2 in Figs. 1–3. This observation is also consistent with the fact that $'''$ characterizes the Java way of doing resolution because $'''$ has functionality$_{1\frac{1}{2}}$ as a precondition. The approach also worked for encapsulation (see Fig. 9).

7. References and related work

For an interesting study on the semantics of object oriented languages we refer to [1]. For the theory of commuting diagrams and functors (category theory) we refer to [2,5,6]. For a discussion that demonstrates how holes in the Java type systems undermine the security, which was one of the main design aims of the language, see [7] (we do not claim that we have captured the entire Java type system; we are still far from that).

The mechanisms for naming as developed in the present paper fit well to the research direction called 'intentional programming' proposed in [16]. The idea of intentional programming is that future programming languages may have all matters of representation and viewing separated from the essential attributed-tree structure that embodies the program's semantics. Putting our mechanisms for naming in this perspective, they make it possible to separate the mechanisms for naming from the other aspects of the language. In that case a user could choose for a particular project whether the program will be developed with all symbol tables being functional$_1$ or if they are to be functional$_{1\frac{1}{2}}$. Viewing tools could translate from one view to another.

In [14] the type system of a subset of Java called BALI is investigated using a representation of lookup tables as partial functions; several operations on lookup tables are defined such as pointwise update $\tau[a := b]$ (which corresponds to our $\{\langle a, b\rangle\} \dashv \tau$), extension of one table by another $\tau_1 \oplus \tau_2$ (which corresponds to our $\tau_2 \dashv \tau_1$) and an operator for hiding.

In [11] a special kind of functors called 'mixins' is developed. Mixins describe the transformation of a class by adding and overriding variables and methods.

Acknowledgements

We like to thank Jan Bergstra, Frank van der Linden, André Engels and Michel Reniers for their support and help.

References

[1] M. Abadi, L. Cardelli, A Theory of Objects, Springer Monographs in Computer Science, Springer, Berlin, 1998.
[2] M.A. Arbib, E.G. Manes, Arrows, Structures and Functors, the Categorial Imperative, Academic Press, New York, 1975.
[3] K. Arnold, J. Gosling, The Java™ Programming Language, Addison-Wesley, Reading, MA, 1996, ISBN 0-201-63455-4.
[4] H.P. Barendregt, The lambda-calculus, its syntax and semantics, Studies in Logic and the Foundations of Mathematics, revised edition, Vol. 103, Elsevier, North-Holland, Amsterdam, 1984.
[5] T.S. Blyth, Categories, Longman, New York, 1986.
[6] R.L. Crole, Categories for Types, Cambridge University Press, Cambridge, 1993.
[7] D.E. Denning P.J. Denning, Internet Besieged, ACM, New York, Addison-Wesley, Reading, MA, 1998.
[8] L.M.G. Feijs, M. Huizer, TSF, a test specification language in: Proceedings of ACP95, CSN report Eindhoven University of Technology, 1995.
[9] L.M.G. Feijs, H.B.M. Jonkers, Formal specification and Design, Cambridge University Press, Cambridge, 1992.
[10] L.M.G. Feijs, H.B.M. Jonkers, C.A. Middelburg, Notations for Software Design, Springer FACIT Series, Springer, Berlin, 1994.
[11] M. Flatt, S. Krishnamurthi, M. Felleizen, Classes and mixins, Conference record of 25th Principles of Programming Languages POPL'98, ACM, New York, 1998, pp. 171–183.
[12] J.R. Hindley, J. Seldin, Introduction to combinators and λ-calculus, London mathematical society student texts 1, Cambridge University Press, Cambridge, 1986.
[13] C.P.J. Koymans, G.R. Renardel de Lavalette, The logic MPL_{ω}, in: M. Wirsing, J.A. Bergstra (Eds.), Algebraic Methods: Theory, Tools and Applications, Springer Lecture Notes in Computer Science, Vol. 394, Springer, Berlin, 1989, pp. 247–282.
[14] T. Nipkow, D. Oheimb, $\mathrm{Java}_{\mathrm{light}}$ is type-safe – definitely, Conference record of 25th Principles of Programming Languages POPL'98, ACM, New York, 1998, pp. 161–170.
[15] M. Schönfinkel, Über die Bausteine der mathematischen Logik, Math. Annalen 92 (1924) 305–316.
[16] C. Simonyi, Intentional Programming – Innovation in the Legacy Age, Presented at the 49th IFIP WG 2.1 meeting, Rancho Santa-Fe, 1996, Internet http://www.research.microsoft.com/research/ip/ifipwg/ifipwg.htm.